SOME ALGORITHM DESIGN METHODS

Steve Tappel

Systems Control, Inc., 1801 Page Mill Road
Palo Alto, California 94304
and Computer Science Department, Stanford University

Abstract

Algorithm design may be defined as the task of finding an efficient data and control structure that implements a given input-output specification. This paper describes a methodology for control structure design, applicable to combinatorial algorithms involving search or minimization. The methodology includes an abstract process representation based on generators, constraints, mappings and orderings, and a set of plans and transformations by which to obtain an efficient algorithm. As an example, the derivation of a shortest-path algorithm is shown. The methods have been developed with automatic programming systems in mind, but should also be useful to human programmers.

I. Introduction

The general goal of automatic programming research is to find methods for constructing efficient implementations of high-level program specifications. (Conventional compilers embody such methods to a very limited extent.) This paper describes some methods for the design of efficient control structures, within a stepwise refinement paradigm. In stepwise refinement (see for instance [1,2]), we view the program specification itself as an algorithm, albeit a very inefficient one. Through a chain of transformation steps, we seek to obtain an efficient algorithm.

Specification → Alg → ... → Algorithm

Each transformation step preserves input-output equivalence, so the final algorithm requires no additional verification.

Algorithm design is a difficult artificial intelligence task involving representation and planning issues. First, in reasoning about a complicated object like an algorithm it is essential to divide it into parts that interact in relatively simple ways. We have chosen asynchronous processes, communicating via data channels, as an appropriate representation for algorithms. Details are in Section II. Second, to avoid blind search each design step must be clearly motivated, which in practice requires organization of the transformation sequence according to high-level plans. An outline of the plans and transformations we have developed is given in Section III, followed in Section IV by the sample derivation of a shortest path algorithm. Sections V and VI discuss extensions and conclude.

This methodology is intended for eventual implementation within the CHI program synthesis system [3], which is under development at Systems Control Inc.

This research is supported in part by the Defense Advanced Research Projects Agency under DARPA Order 3687, Contract N00014-79-C-0127, which is monitored by the Office of Naval Research. The views and conclusions contained in this paper are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of SCI, DARPA, ONR or the US Government.

II. Process graph representation of algorithms

Our choice of representation is motivated largely by our concentration on the earlier phases of algorithm design, in which global restructurings of the algorithm take place. Most data structure decisions can be safely left for a later phase, so we consider only simple, abstract data types like sets, sequences and relations. More importantly, we observe that conventional high-level languages impose a linear order on computations which is irrelevant to the structure of many algorithms and in other cases forces a premature committment to a particular algorithm. To avoid this problem, we have chosen a communicating process representation in which each process is a node in a directed graph and processes communicate by sending data items along the edges which act as FIFO queues. Cycles are common and correspond to loops in a conventional language.

The use of generators (or producers) in algorithm design was suggested by [4]. Our representation is essentially a specialized version of the language for process networks described in [5]. Rather than strive for a general programming language we use only a small set of process types, chosen so that: (1) the specifications and algorithms we wish to deal with are compactly represented, and (2) plans and transformations can be expressed in terms of adding, deleting or moving process nodes. The four process types are:

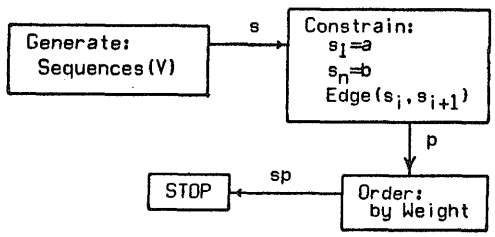
Generator: produces elements one by one on its output edge.
Constraint: acts as a filter; elements that satisfy the constraint pass through.
Mapping: takes each input element and produces some function of it. If the function value is a set its elements are produced one by one.
Ordering: permutes its input elements and produces them in the specified order.

The representation is recursive, a very important property. There can be generators of constraints, constraints on constraints, mappings producing generators, etc. Most of the same design methods will apply to these "meta-processes".

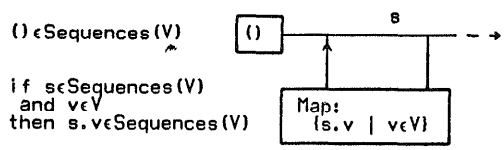
To illustrate the representation, we encode the specification for our sample problem of finding the shortest path from a to b in a graph.

Notation and terminology for shortest path. A directed graph is defined by a finite vertex set V and a binary relation Edge(u,v). A path p is a sequence of vertices $(p_1 \ldots p_n)$, in which $Edge(p_i,p_{i+1})$ holds for each pair. The "." operator is used to construct sequences: (u ... v).w = (u ... v w). Every edge of the graph is labelled with a positive weight W(u,v) and the weight of an entire path is then $Weight(p) = W(p_1,p_2)+\ldots+W(p_{n-1},p_n)$. The shortest path from a to b is just the one that minimizes Weight.

A specification should be as simple as possible to ensure correctness. Shortest path can be simply specified as: generate all paths from a to b, and select the shortest. We express selection of the shortest path in a rather odd way, feeding all the paths into an ordering process whose very first output will cause the algorithm to stop. The point is that by using a full ordering for this comparatively minor task, we can apply all the plans and transformations for orderings. As for the paths from a to b, they are defined as a certain kind of sequence of vertices, so we introduce a generator of all vertex sequences and place constraints after it to eliminate non-paths. This completes the specification.



Selection of an appropriate internal structure for the generator of Sequences(V) is actually part of the design process, but to simplify the example we will take as a default the usual recursive definition of sequences. The recursion in the definition corresponds to a cycle in the process graph.



The generation process starts when the empty sequence () is produced on the "s" edge. From the "s" edge it goes to the constraint and also to the mapping, which produces the set of all one-vertex sequences ().v, for v∈V. These are fed back to generate two-vertex sequences, and so on. A mapping cycle like this is a very common kind of generator.

III. Methods for algorithm design

The program specification from which design starts is typically written as an exhaustive generate-and-test (or generate-and-minimize) process, and bears little resemblance to the algorithm it will become. The design methods all have the goal of incorporating constraints, orderings or mappings into the generator, or else the goal of planning or preparing to do so. To incorporate a constraint means to modify the generator so that it only generates items which already satisfy the constraint; to incorporate an ordering means to modify the generator so it generates elements directly in that order; and to incorporate a mapping f means to generate elements f(x) instead of elements x.

Accordingly, the methods fall into three main classes, briefly described below. Superimposed upon this class division is a heirarchy (not strict) with multi-step plans at the higher levels and a large number of specific syntactic transformations at the bottom. The heirarchy is organized according to goals and subgoals. Heuristics and deduction rules are required to support the planning activity. At the time of writing, a total of about 20 methods have been formulated not counting low-level syntactic transformations.

Constraint methods. The goal of constraint methods is to reduce the number of elements generated. The top level plan for constraints says to:

1. *propagate* constraints through the process graph to bring them adjacent to a generator,
2. *incorporate* constraints into a generator whenever possible, and if the results are not satisfactory,
3. *deduce* new constraints beyond those given in the specification, and repeat.

Each of the three subtasks is nontrivial in itself and is carried out according to a set of (roughly speaking) intermediate-level methods. For (2), an intermediate-level method that we use several times in the Shortest Path derivation is:

The constraint incorporation plan ConstrainComponent. ConstrainComponent applies when a constraint on composite objects x (sets, sequences, not numbers) is reducible to a constraint on a single component c of x, i.e. $P(x) \equiv P'(x_c)$. ConstrainComponent then gives a plan:

1. Inside the generator, find the sub-generator of values for component c. If necessary, manipulate the process graph to isolate this generator. Again, other methods must be called upon.
2. Remove constraint P and add constraint P' to the sub-generator.

Ordering methods. Another group of methods is concerned with the deduction, propagation and incorporation of orderings on a generated set. These methods are analogous to the methods for constraints but more complicated. In the Shortest Path derivation we use a powerful transformation, explained rather sketchily here:

The ordering incorporation transformation InterleaveOrder. InterleaveOrder applies when an ordering R is adjacent to a generator consisting of a mapping cycle, in which the mapping f has the property R(x f(x)) for all x. In other words, f(x) is greater than x under the ordering R. InterleaveOrder moves the ordering inside the mapping cycle and adds a synchronization signal to make the ordering and mapping operate as coroutines. The ordering produces an element x, the mapping receives it and produces its successors f(x) (there would be no need for the ordering at all if f were single-valued), then the ordering produces the next element and so on.

Mapping methods. The methods for incorporating a mapping into a generator are mostly based upon recent work in the "formal differentiation of algorithms" [6] and are related to the well-known optimizing technique of reduction in operator strength. (They are not used in our sample design.)

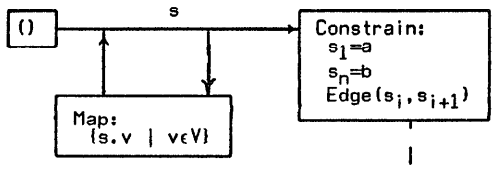
Some syntactic transformations and other methods not described in this section will appear in the derivation.

IV. Example: Design of a shortest path algorithm

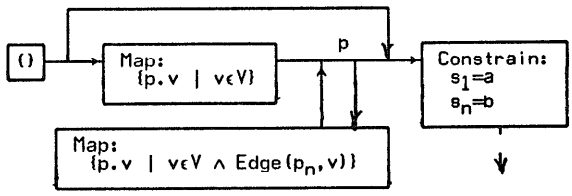
In the design which follows, the specification will be transformed from an inefficient generate-and-minimize scheme into a dynamic programming algorithm. The final algorithm grows paths out from vertex a, extending only the shortest path to each intermediate vertex, until reaching b. Of necessity we omit many details of the design.

IV.1. Constraint methods.

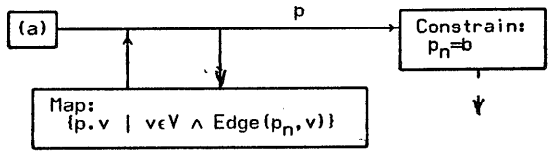
Since the specification's constraints are already next to the generator (step 1), the overall plan for constraints says to try to incorporate them (step 2.) We will follow the heuristic of incorporating the strongest constraint first. Right now, the algorithm reads



Incorporate the Edge constraint. More detail will be shown in this first step than in later derivation steps. ConstrainComponent applies because once a vertex s_i has been added to a sequence, the constraint Edge(s_i,s_{i+1}) reduces to a constraint on the single component s_{i+1}. (This reasoning step is really the application of another method, not described here.) Step (1) in the ConstrainComponent plan says to find the generator of values for components s_{i+1}. Though we have written it in linear form for convenience, the expression {s.v | v∈V} is really a generator followed by a mapping. Unfortunately "v∈V" generates s_1 as well as the desired s_{i+1} values, so we have to unroll one cycle of the graph to isolate the generator of s_{i+1} values. (Again, we have applied methods not described in this paper.) Step (2) is now possible and consists in constraining v to satisfy Edge(s_n,v). With the Edge constraint incorporated, only paths are now being generated so we change s to p in the diagram.



Incorporate the constraint that p_1=a. Since the p_1=a constraint refers only to a component of p, ConstrainComponent applies again. We constrain v in the first "v∈V" generator to be equal to a. After simplifying, we obtain



Incorporate the constraint that p_n=b. Once again ConstrainComponent applies. This time, however, we are unable to isolate a generator for the last vertex of paths. The last vertex of one path is the next-to-last vertex of another, and so on. ConstrainComponent fails, other methods fail too; we leave the p_n=b constraint unincorporated.

Deduce new constraint. In accordance with the general constraint plan (step 3) we now try to deduce more constraints. One method for deducing new constraints asks: do certain of the generated elements have *no effect whatsoever* upon the result of the algorithm? If the answer is "yes", try to find a predicate that is false on the useless elements, true on others. Motive: if we later succeed in incorporating this constraint into the generator, the useless elements will never be produced.

Now consider the Order + STOP combination. Because all it does is select the shortest path, any path which is *not* shortest will have no effect! The corresponding constraint says:

p is a shortest path from a to b.

A further deduction gives the even stronger constraint that *every subpath* of p must be a shortest path (between its endpoints). Incorporation of this constraint is complex and is deferred till after incorporation of the Weight ordering.

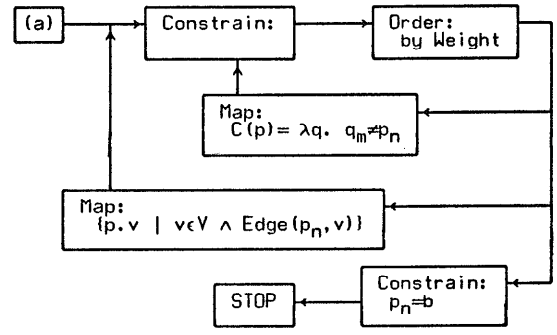
IV.2. Ordering methods.

So far paths are generated according to the partial order of path inclusion; path p is generated before path q if q = p.u .. .v for some vertices u,..,v. We may generate a lot of paths to b before generating the shortest one - possibly an infinite number. However if the Weight ordering can be incorporated into the path generator, then only a single path to b (the shortest one) will ever be generated.

Propagate Ordering. Before applying an incorporation method we need to bring the Weight ordering next to the generator. Constraints and orderings commute so this is easy.

Incorporate the ordering into the generator. The InterleaveOrder method applies, because Weight(p.v) is greater than Weight(p). It moves the ordering from outside the generator cycle to inside and also causes the ordering to wait for the mapping to finish extending the previous path before it produces another.

Incorporate new constraint. The "p is a shortest path" constraint is readily incorporated now: the shortest path to any vertex will be the *first* path to that vertex. Any later path q, with the same last vertex q_m=p_n, can be eliminated by a new constraint C(p) = λq. q_m=p_n. We introduce a mapping to produce these new constraints C(p), and now we have a *generator* of *constraints*. The result of the last three steps is



The algorithm is now a breadth-first search for a path to b, with elimination of non-optimal paths at every vertex. Despite various inefficiencies that remain, the essential structure of a dynamic programming algorithm is present. One interesting improvement comes from incorporating the generated constraints C(p) into the generator of paths, using ConstrainComponent. To complete the derivation would require data structure selection and finally a translation into some conventional programming language.

V. Other results and limitations

Besides the Shortest Path algorithm shown here (and variants of it) the algorithm design methods have been used [7] to derive a simple maximum finding algorithm and several variants on prime finding including the Sieve of Eratosthenes and a more sophisticated linear-time algorithm. In these additional derivations, no new process types and only a few new methods had to be used. Single and multiple processor implementations have informally been obtained from process graph algorithms, for both prime finding and Shortest Path.

More algorithms need to be tried before specific claims about generality can be made. The intended domain of application is combinatorial algorithms, especially those naturally specified as an exhaustive search (possibly over an infinite space) for objects meeting some stated criteria, which can include being minimal with respect to a defined ordering. Backtrack algorithms, sieves, many graph algorithms and others are of this kind [8].

The methods described here are quite narrow in the sense that a practical automatic programming system would have to combine them with knowledge of:

1. Standard generators for different kinds of objects. Our methods can only modify an existing generator, not design one.
2. Data structure selection and basic operations such as searching a list.
3. Efficiency analysis to determine if an incorporation really gives a speedup.
4. Domain specific facts, e.g., about divisibility if designing a prime finding algorithm.
5. How to carry out the final mapping of process graph into a conventional programming (or multiprogramming) language.

VI. Discussion and Conclusions

The main lesson to be learned from this work is the importance of using an abstract and modular representation of programs during algorithm design. Details of data structure, low-level operations and computation sequencing should be avoided, if possible, until the basic algorithm has been obtained. (Since some algorithms depend crucially upon a well-chosen data structure, this will not always be possible.) Further, it is advantageous to represent algorithms in terms of a small number of standard kinds of process, for which a relatively large number of design methods will exist. The results so far indicate that just four standard processes suffice to encode a moderate range of different specifications and algorithms. Presumably others will be required as the range is extended, and it is an important question whether (or how long) the number can be kept small. A similar question can be asked about the design methods.

One would not expect methods based upon such general constructs as generators, constraints, orderings and mappings to have much power for the derivation of worthwhile algorithms. For instance, if we had explicitly invoked the idea of dynamic programming, our derivation of a shortest path algorithm would have been shorter. For really difficult algorithms, the general methods may be of little use by themselves. We suggest that they should still serve as a useful complement to more specific methods, by finding speedups (based on incorporation of whatever constraints, orderings and mappings may be present) in an algorithm obtained by the specific methods.

As a final issue, it is interesting to speculate how the stepwise refinement approach to programming might be used by human programmers. Use of a standard set of process types and correctness-preserving transformations would be analogous to the formal manipulations one performs in solving integrals or other equations. If that were too restrictive, perhaps one could use the methods as a guide, without attempting to maintain strict correctness. After obtaining a good algorithm, one could review and complete the design, checking correctness of each transformation step. The result would be a formally correct but also well-motivated derivation.

Acknowledgements.

Many helpful ideas and criticisms were provided by Cordell Green, Elaine Kant, Jorge Phillips, Bernard Mont-Reynaud, Steve Westfold, Tom Pressburger and Sue Angebranndt. Thanks also to Bob Floyd for sharing his insights on algorithm design.

References

1. Balzer, Robert; Goldman, Neil; and Wile, David. "On the transformational implementation approach to programming", *Proc. 2nd Int'l Conference on Software Engineering* (1976) 337-344.
2. Barstow, David R. *Knowledge Based Program Construction*, Elsevier North-Holland, New York, 1979.
3. Phillips, Jorge and Green, Cordell. "Towards Self-Described Programming Environments", Technical Report, Computer Science Dept., Systems Control, Inc., Palo Alto, California, April 1980.
4. Green, Cordell and Barstow, David R. "On Program Synthesis Knowledge", *Artificial Intelligence*, 10:3 (1978) 241-279.
5. Kahn, Gilles and MacQueen, David B. "Coroutines and Networks of Parallel Processes", *Information Processing 77*, IFIP, North-Holland Publishing Company, Amsterdam, (1979) 993-998.
6. Paige, Robert. "Expression Continuity and the Formal Differentiation of Algorithms", Courant Computer Science Report #5, (1979) 269-658.
7. Tappel, Steve. "Algorithm Design: a representation and methodology for control structure synthesis", Technical Report, Computer Science Dept., Systems Control, Inc., Palo Alto, California, August 1980.
8. Reingold, Edward M., Nievergelt, Jurg, and Deo, Narsingh. *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1977.
9. Elschlager, Robert and Phillips, Jorge. "Automatic Programming" (a section of the *Handbook of Artificial Intelligence*, edited by Avron Barr and Edward A. Feigenbaum), Stanford Computer Science Dept., Technical Report 758, 1979.
10. Floyd, R. W. "The Paradigms of Programming" (Turing Award Lecture), *CACM* 22:8 (1979) 455-460.